High-performance end-user analysis in pure Julia programming language

Jerry Ling[1,*] and *Tamás* Gál[2,**]

[1]Harvard University
[2]Erlangen Centre for Astroparticle Physics

Abstract. We present tools for high-performance analysis written in pure Julia, a just-in-time (JIT) compiled dynamic programming language with a high-level syntax and performance. The packages we present center around `UnROOT.jl`, a pure Julia ROOT file I/O package that is optimized for speed, lazy reading, flexibility and thread safety.
We discuss what affects performance in Julia, the challenges, and their solutions during the development of `UnROOT.jl`. We highlight type stability as a challenge and discuss its implication whenever any "compilation" happens (incl. Numba, Jax, C++) as well as Julia's specific ones.
We demonstrate the performance and "easy to use" claim by comparing `UnROOT.jl` against popular alternatives (RDataFrame, Uproot, etc.) in medium-size realistic benchmarks, comparing both performance and code complexity.
Finally, we also showcase real ATLAS analysis workflows both locally and on an HPC system, highlighting the composability of `UnROOT.jl` with multi-thread/process and out-of-core distributed computing libraries.

1 Introduction

There are many reasons why the Julia programming language is a good fit for scientific computing in general, but also specifically for high-energy physics (HEP). In this paper, we focus on the aspects of Julia that impact end-user analysis workflows such as performance, multi-threading, while skip other features such as reproducibility, binary dependencies handling etc.

To read about complete discussions with comparisons and case studies on Julia's potential adoption in wide HEP community, we refer readers to the recent white paper [1].

2 Paper structure

This paper is structured in a user-driven way: in each of the two sections, we explore what end-users want in a typical HEP data analysis task, and see how Julia the ecosystem and the `UnROOT.jl` package we developed meet those needs.

*e-mail: jling@g.harvard.edu
**e-mail: tamas.gal@fau.de

In the first section, we focus on single-node computing, since HEP end-user analysis often only involves naive parallelism. We briefly explore Julia's compilation model to explain why Julia can be "easy to write" while having "no performance compromise".

In the second section, we demonstrate how the same code can be re-used easily to scale to multi-node high-performance computing/high-throughput computing (HPC/HTC) infrastructures that are already in place for HEP applications (e.g. HTCondor).

3 Source of Julia's performance and `UnROOT.jl`

We commonly think **performance** and **expressiveness** as a trade-off when it comes to programming languages. Typical languages we use to achieve full hardware potential includes languages such as C++; while less verbose and thus easier to write languages such as Python are less performant.

One of the fundamental reasons contributing to both performance and apparent verbosity lies in the *type* information available to the compiler. The more information the compiler has, the better it can emit most optimized machine instructions given user-defined tasks.[1]

In C++, the *type* information (e.g. variable types, function arguments types, function return types) are usually spelled out in the source code, which increases verbosity, but serves as a sure way to pass the information to the compiler.

Julia[2], as a language designed around JIT compilation, employs "specialized compilation" [3] to get the benefits from both worlds: speed and ease of use. We look at a simple example to demonstrate how Julia's compilation model works:

```julia
julia> function mysum(ary)
           s = zero(eltype(ary))
           for x in ary
               s += x
           end
           return s
       end
```

This function can technically be passed any argument, as long as function calls such as `eltype` are also well-defined for the input variable. When looking at this code alone, it's hard to guess how Julia can achieve peak performance, for example, how does it know what CPU instruction the plus (+) should correspond to?

The trick is to compile a specialized instance of the function, for different types of input arguments. Because Julia uses JIT compilation, it's easy to see the compilation in action. After we defined the function but before it was ever called, we can see there are 0 instances compiled:

```julia
julia> using MethodAnalysis

julia> methodinstances(mysum)
[] # collection of compiled instances is empty
```

Now, if we execute the function first on a collection of integers and then on a collection of floats, you can see new instances of the function gradually being compiled:

[1]Python's reference implementation uses an interpreter which makes the comparison less direct but the overall idea stands even if you switch to Cython, PyPy, or Numba.

```
julia> mysum([1, 2, 3])
6

julia> methodinstances(mysum)
1-element:
 MethodInstance for mysum(::Vector{Int64})

julia> mysum([1.0, 2.0, 3.0])
6.0

julia> methodinstances(mysum)
2-element:
 MethodInstance for mysum(::Vector{Int64})
 MethodInstance for mysum(::Vector{Float64})
```

The compiled instance will get cached and won't add extra cost the next time this function is called on known input types. This is the main mechanism in Julia to achieve both speed and expressiveness.

4 UnROOT.jl performance and multi-threading

Given the high-throughput nature of most end-user analysis computing, a big common bottleneck is processing the data stored in .ROOT files (specifically, the TTree container, and in the future, RNTuple container).

We developed UnROOT.jl [4] to address this demand. UnROOT.jl is a pure Julia ROOT file reader that focuses on ease of use, speed, and thread safety. Here is a simple example of an event loop in Julia:

```
    using UnROOT
    tree = LazyTree("./data.root", "Events")
    for evt in tree
        muon_HT = sum(evt.Muon_pt)
        if muon_HT < 200
            continue
        end
        #...
    end
```

Following our discussion on the relationship between type information and performance, we can see that the compiler might have difficulties inferring the type of evt.Muon_pt when trying to compile the loop body. The workaround is to encode the name and element type of the branch in the type of evt. Here is an example of name-type encoding in the built-in NamedTuple:

```
julia> x = (a=1, b=2)

julia> typeof(x)
NamedTuple{(:a, :b), Tuple{Int64, Int64}}
```

This is done by UnROOT.jl at parsing time, end-users can simply write the for loop without any type annotation.

UnROOT.jl is also very amenable to multi-threading:

```julia
using UnROOT
tree = LazyTree("./data.root", "Events")
Threads.@threads for evt in tree
    muon_HT = sum(evt.Muon_pt)
    if muon_HT < 200
        continue
    end
    #...
end
```

The built-in @threads macro will automatically distribute the events in true memory-shared multi-threading fashion (think OpenMP). End-users, however, are not confined to this primitive parallel construct, they can use any multi-threading library they like, such as ThreadsX.jl[2].

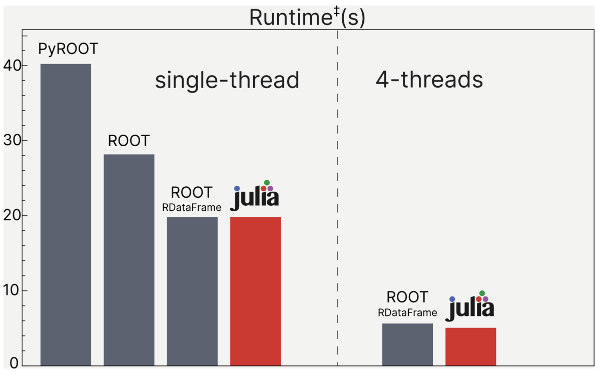

Figure 1. Sythetic benchmark of UnROOT.jl with source code available.

Finally, it is worth mentioning that the alternative “vector-style” operation is still available since each column of the LazyTree conforms with AbstractVector interface.

5 Out-of-core parallel computing

Although Julia’s multi-threading is very powerful, the throughput of a single node is still limited, and users often encounter difficulties when trying to scale their analysis to a cluster of nodes. The usual problems are: 1) debugging different node architectures and environments; 2) long turn-around time for job submission when debugging.

Here we demonstrate how Julia’s built-in out-of-core parallel computing library can be used even in a very primitive setup to scale an interactive analysis to a cluster of nodes.

First, adding workers and execute arbitrary code on them is as simple as:

[2]https://github.com/tkf/ThreadsX.jl

‡https://github.com/Moelf/UnROOT_RDataFrame_MiniBenchmark

```
# [local code working!]
julia> using ClusterManagers, Distributed, Revise

julia> addprocs(HTCManager(4))
# Waiting for 4 workers: 1 2 3 4 .

julia> @fetchfrom 1 gethostname()
"login02.af.uchicago.edu" # <--- user's login node

julia> @fetchfrom 2 gethostname()
"c028.af.uchicago.edu" # <--- a HTCondor node
```

Here we have access to the HTCondor cluster at University of Chicago, but the same idea applies to any HTCondor setup. And ClusterManagers.jl also supports other cluster managers such as Slurm.

Now the user can load their analysis code everywhere at once:

```
julia> @everywhere using WVZAnalysis

julia> run_analysis(..)
```

Inside the `run_analysis` function, we should assume that users used functions such as `pmap()` etc. which will automatically utilize all the workers that can be reached by `Distributed.workers()`. Furthermore, in the event that user needs to debug their analysis, they can simply edit the code, and the changes will be propagated to all the workers by `Revise.jl`, and only the changed code path will be recompiled.

It is hard to find a dedicated cluster to conduct a full benchmark, but preliminary tests show that the scaling is linear in the number of nodes since the analysis is embarrassingly parallel:

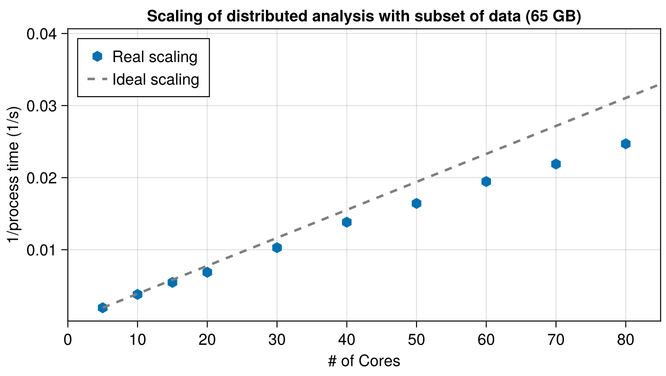

Figure 2. Scaling as a function of number of nodes. Expected to be linear, falling off due to I/O at cluster site.

6 Conclusion

In this paper, we briefly demonstrated Julia's compilation model and why it hits the sweet spot between ease of use and performance. We also demonstrated how `UnROOT.jl` is implemented to maximally utilize Julia's compilation model in order to achieve high performance for typical HEP end-user analyses. We also demonstrated how the same code can be re-used to scale to a cluster in a real analysis workflow, which is currently being used by one of the authors in ATLAS analysis.

References

[1] J. Eschle, T. Gal, M. Giordano, P. Gras, B. Hegner, L. Heinrich, U.H. Acosta, S. Kluth, J. Ling, P. Mato et al., *Potential of the julia programming language for high energy physics computing* (2023), `2306.03675`

[2] J. Bezanson, A. Edelman, S. Karpinski, V.B. Shah, SIAM review **59**, 65 (2017)

[3] A. Pelenitsyn, J. Belyakova, B. Chung, R. Tate, J. Vitek, Proc. ACM Program. Lang. **5** (2021)

[4] T. Gál, J. Ling, N. Amin, *UnROOT: an I/O library for the CERN ROOT file format written in Julia* (2021), `https://github.com/JuliaHEP/UnROOT.jl/`